PARALLEL DATA TRANSFER OVER MULTIPLE CHANNELS WITH DATA ORDER PRIORITIZATION

[0001] This application claims priority from U.S. Provisional Application Serial No. 60/451,295, filed February 28, 2003, the entire content of which is incorporated herein by reference.

TECHNICAL FIELD

[0002] The invention relates to computer networks and, in particular, to the transfer of data over computer networks.

BACKGROUND

[0003] With the increasing adoption of rich-media applications involving audio and video data, and the growing adoption of broadband internet connections, the characteristics of network file transfers are quickly changing. While only a few years ago small data objects, such as HTML documents, electronic mail (e-mail), and images, dominated network traffic, there is currently an explosion in the use of rich-media technologies, such as streaming audio and video. Even the smallest video files are often hundreds of times larger than a typical e-mail or word processing document, as illustrated in FIG. 1.

[0004] With gigabyte-sized files now becoming a reality for many applications and file transfers taking many hours, even on a broadband connection, there exists a need to optimize the delivery of such files and make that delivery robust in the face of changing network conditions.

[0005] Another trend that is motivating the need for data transfer optimization is that end-user broadband connections are quickly becoming as fast as the web servers from which they are downloading the content. This means that increasingly, the originating server is the bottleneck of the data transfer and not the receiver. Conventional systems referred to as "Content Delivery Networks", such as that from Akamai Technologies, Inc., of Cambridge, MA, work to avoid this problem by selecting a server close to the user in order to provide them with a fast download. Even with this technology, however, there can still be a bottleneck between the chosen server and the receiving host.

BRIEF DESCRIPTION OF DRAWINGS

[0006] FIG. 1 is a graph illustrating typical file sizes for different types of files.

[0007] FIG. 2 is a diagram illustrating parallel data transfer over multiple channels.

[0008] FIG. 3 is a diagram illustrating the manner in which transfer rates are increased by using multiple channels.

[0009] FIG. 4 is a block diagram illustrating an exemplary software architecture for implementing the principles of the invention.

[0010] FIG. 5 is a flow diagram illustrating techniques for prioritizing data for transferring data over multiple channels.

DETAILED DESCRIPTION

[0011] In general, the invention is directed to techniques for transferring data over multiple channels in parallel, possibly from multiple servers. This allows the bandwidth from the various sources to be aggregated to provide a very fast download.

[0012] As illustrated in FIG. 2, a file may be split into multiple pieces across the N servers, e.g., four servers, and may be transferred from all of them in parallel. Thus, even though each server only has a capacity of 90KB/s, the total transfer rate is 360 KB/s.

[0013] The delivered file may be split up in a number of ways to deliver it from multiple servers in parallel. One approach, for example, is to split the file into a number of fixed-sized blocks and transfer one block from each server. For example, as illustrated in FIG. 2, four servers may be utilized, and a 20MB file may be split into 4 pieces. Thus, each server may transfer 5MB of data to the receiver. However, an individual server may be slow or unavailable, and thus would adversely affect the download speed.

[0014] An enhancement to the first approach is to split the file into N pieces (P1, P2, …, Pn). The receiver would request P1 from server1, P2 from server2, P3 from server3, and P4 from server4. Once one of those pieces is received, the receiver will request the next desired piece from the available server.

[0015] One of the unique attributes of the invention is that it allows prioritization of the order in which data is received during a parallel data transfer. Two common ways in which data is accessed within a file is either "sequential" or "random access" order. Most network protocols, such as http and FTP, transfer data in sequential order whereby the first byte of the file is received first, the second byte is received second, etc until the end of the file is reached. Most network-aware applications, such as web browsers and computer video and audio players, expect content to be received in sequential order and can actually play back content while it is being downloaded if the data is provided to the media player in sequential order.

[0016] Another increasingly common scenario that motivates data transfer optimization is the problem of moving very large files over long distances, such as between sites on two different continents. Today's standard protocols for data transfer are FTP and http, both of which operate on top of the TCP protocol. TCP contains features to provide reliable transfer of data by acknowledging the receipt of data packets back to the sender. In normal Internet operation, TCP performs very well and is quite reliable. However, TCP's performance can degrade considerably over "Long, Fat Networks" (LFNs) – networks with very high latency while having very high bandwidth. Long Fat Networks include high speed intercontinental networks and satellite-based networks. By default, TCP's buffers are not large enough to fill the capacity of a LFN, often only providing 10% of the possible speed available over a long fat network.

[0017] Parallel data transfer can also be used to improve this situation. In this scenario there is a single server and a single client, but multiple "channels" are established over the route between client and server. By transferring data along multiple channels in parallel, transfer rates up to 10x faster than normal http or FTP can be achieved, as illustrated by FIG. 3.

[0018] The invention provides reliable, high throughput, and low latency data transfer by downloading data from multiple sources or channels in parallel. The software is optimized for both very large and very small content and is uniquely designed to provide high-speed parallel streaming. The invention provides reliable downloads even when the sources have poor connectivity, and is thus ideal for distributing data over unreliable networks such as wireless networks or peer-to-peer networks. Some advantages that may be achieved by the invention are as follows:

- Speed – It includes optimizations such as Dynamic Range Requests, Pipelining, and Range Preemption to download the data as absolutely fast as possible.

- Intelligence – The engine will automatically discover and download from the most desirable mirrors or sources. It constantly monitors and responses to changing network conditions, optimizing the download on the fly.
- Fault Tolerance – The majority of software is not robust in the face of network problems, they hit a snag and immediately throw an error or hang. In contrast, the invention automatically compensates for network problems when they arise. It routes around problems, retries, and will faithfully deliver the data.
- Streaming – The engine was specifically designed to provide the fastest progressive downloads. Using the technology, one can actually watch a video as it is being downloaded from multiple sources in parallel. This is perfect for Video-On-Demand, P2P, and distance learning applications.
- Security – Full support for encryption via SSL/TLS, including the ability to download content from "https" URLs.
- Streaming Integrity Checking – The invention supports various integrity checking mechanisms, including "Merkle Hash Trees" which can verify the integrity of data in a streaming fashion.
- Corruption Repair – The invention can automatically detect and repair corruption for any file being verified by the integrity checking system. Together these features enable "Self-Healing Downloads".

[0019] The details of one or more embodiments of the invention are set forth in the accompanying drawings and the description below. Other features, objects, and advantages of the invention will be apparent from the description and drawings, and from the claims.

[0020] FIG. 4 illustrates one exemplary software architecture for implementing the principles of the invention. More specifically, the software architecture includes a number of cooperating software modules. The embodiments of the exemplary software architecture may reside on each of the N servers originating a data transfer to a client device. In particular, portions or all of the software modules of FIG. 4 may be loaded on each source server.

Channels

[0021] The idea of having multiple "channels" from which data can be received is fundamental to the invention. A channel is any communication path along which information can be transferred. A bit is the most basic unit of information, but the invention can work equally well with any unit of information, such as a word, or a megabyte. It is anticipated that for the purposes of the invention, a byte or eight bits will be the basic unit.

[0022] While the invention is most specifically designed for transferring data across computer networks, the invention does not require the channels used to be of the same type. Channels may be provided by various networking protocols such as FTP, HTTP, or a peer-to-peer application speaking a custom protocol. Channels could also include data coming from a CD-ROM, hard disk storage, or temporary cache storage. Additional examples of channels could include wireless networks or high-speed interconnects such as USB, Firewire, or Fiber Channel.

[0023] One suggested way to implement the invention is to identify a list of available channel sources from which the content can be received. Using a Channel Connector, a channel may be established to a source and data may be received from that channel. The inventors suggest identifying channel sources by a Uniform Resource Indicator (URI). URIs are a superset of identifiers that, among other things, cover Uniform Resource Locators (URLs). Examples of URLs include (http://microsoft.com/) and (ftp://ftp.cdrom.com/) which use the HTTP and FTP protocols respectively to connect to channels and retrieve content.

[0024] The invention allows any number of channels to be opened to any combination of sources. For the case of optimizing data transfer over Long Fat Networks, the invention will utilize multiple simultaneous channels to a single source. In the case where it is desired to aggregate bandwidth from multiple sources, then multiple channels may be opened to multiple sources. Also note, that two channels to a single source, may also take different routes, thus making the characteristics of each channel different even though they may share the same protocol and source.

Channel Adapters

[0025] Channel Adapters are employed to translate encoded information from the format used to transfer it across the channel to the format desired for the download. Examples of channel encodings would include encryption, compression, parity or error correction coding, and delta or difference encoding.

[0026] For example, the invention can receive data over a compressed channel where the transport protocol supports compression. There would then be a channel adapter that automatically decompresses the encoded content and provides the Channel Receiver with the content in uncompressed form.

[0027] In another example, the invention would receive data over an encrypted channel using a protocol such as the Secure Sockets Layer (SSL) or Transport Layer Security (TLS). There would then be a channel adapter that automatically decrypts the content and provides the Channel Receiver with the content in unencrypted form. Integrity checking can also be performed within the Channel Adapters. An integrity checking error at this level will be signaled as a transport level error and the bytes that were detected to be corrupt will be thrown out and the channel will close with an error.

[0028] It may also be desired to not employ channel adapters when retrieving encoded content. A non-limiting example of this would be a video file wrapped in a Digital Rights Management encryption layer that is decoded by an external media player process.

[0029] Another service that can be added at the Channel Adapter layer is bandwidth throttling services. Various bandwidth throttling policies can be employed so that downloads from certain sources are throttled, or certain combinations of sources are throttled, or the overall download rate is throttled, or aggregate rate across multiple downloaders is throttled. One such policy is to prefer to download from sources with a cheaper bandwidth cost and if those sources are available, throttle the more expensive sources so that fewer bytes are retrieved from them.

[0030] The invention also implements Channel Adapters such that multiple adapters may be combined together so that multiple decodings may be performed on the incoming data on a single channel, such as both decrypting and decompressing content.

Control Interface

[0031] The control interface allows external applications or users to control the behavior of the parallel download system. The control interface provides a number of features:

- Channel sources can be specified before the download starts.
- New sources to be added while the download is in progress.
- The download can be started, stopped, suspended, or resumed.
- External applications can receive notifications of key events, such as when a channel connection occurs, when bytes are received, when bytes are verified, or when bytes are discovered to be corrupt.
- Integrity verifiers can be specified
- Connection verifiers can be specified that include metadata for the data that is desired to be transferred
- Various properties may be set to control the behavior of the downloads, such as whether or not the download should signal an error and abort the download process when the available sources have failed beyond a certain limit. There is also a fail-safe mode that will cause the download to continue until it is either canceled or the download has successfully completed.
- A download in progress may be canceled, an individual connect process may be canceled, or a range download process may be canceled.

[0032] The control interface also allows control over the persistent data storage. If the storage is a regular computer file system, a destination file can be specified ahead of time or the destination can be set during the download.

Output Stream

[0033] The output stream is a special channel that provides the downloaded data to an external application or user. The output stream makes data available as soon as that data has been verified by the integrity verifiers. Normally the output stream will deliver data in a sequential fashion, though it can also provide data in a random access fashion. The mode by which the output stream is accessed can directly affect the data prioritization by having the output stream notify the Prioritization Engine about which bytes are the highest priority. By using the default prioritization policy of optimizing sequential data access, the output stream combined with the parallel data transfer across multiple channels provides the very novel concept of parallel streaming downloads.

[0034] The output stream may deliver data to multiple consuming applications or users. Thus one consumer may be a media player that is watching a video file being downloaded, while the data may be simultaneously streamed to hard disk storage for later playback.

[0035] Since the output stream itself is a data channel, it can be used as a source for other parallel streaming downloads that are taking place, either on the same computer, or on other computers. When the output stream is acting as a source for a download taking place on another computer, this is a peer-to-peer network transfer which provides the very novel concept of a peer-to-peer network with parallel streaming downloads. The invention can also be used to enhance the functionality and performance of *swarming download systems* including those embodied by the Apparatus, Method and System for an Acknowledgement Independent Equalized Data Packet Transfer Mechanism over a Peer to Peer Network for which a non-provisional patent application was filed on December 28, 2001 with the application Serial No. 10/033,305.

Source Scheduler

[0036] The source scheduler has the important job of scheduling which sources will be downloaded from at a given time. The source scheduler keeps track of various information about each channel source and uses that information to decide which sources to download from. As connections are made to the sources and data is downloaded, the channel scheduler will monitor this activity and record various information about the source such as the latency to the source, or how long it takes to receive data, and the current transfer rate from that source.

Source Ranker

[0037] The first step in the source scheduler is to rank the channel sources. Various mechanisms may be used to rank the sources. These mechanisms may include:

- **External Ranking** – The source ranker can use a preconfigured ranking of sources as specified by the user or an administrator. The ranking can also be obtained from an external ranking service that could run on a separate server on the Internet.

- **Throughput** – The sources can be ranked based on their throughput. Faster sources will be given a higher ranking
- **Latency** – The sources can be ranked based on their latency. Sources with the lowest latency will be given a higher ranking.
- **Network Hops** – The sources can be ranked based on the number of network hops to the source. Sources with the lowest hop count will be given a higher ranking.
- **Geographical Location** – The sources can be ranked based on their geographical location relative to the downloader. Closer sources will be given a higher anking.
- **Channel Cost** – The sources can be ranked based on how much it costs to transfer data over that path. Cheaper sources are given a higher ranking.

[0038] The ranking mechanisms can also be combined with any sort of relative weighting between the metrics. One common way to combine metrics with weighting is to use a polynomial with the variables as the metric and the coefficients used as the relative weighting. It is also possible to change the weighting of the various metrics based on external factors, such as the time of day. For instance, during critical business hours, it may be desirable to retrieve data from the fastest possible sources, while during the evening it may be desirable to minimize the cost of the data transfer.

[0039] If the goal is purely to provide the fastest possible data transfer, then a combination of the throughput and latency metrics works quite well in practice for ranking the various sources.

Slot Manager

[0040] The basic strategy that the source scheduler employs is to assign channel sources to a number of source "slots". The source scheduler is configured with an average number of slots it should maintain as well as a maximum number of slots. During a download channels are frequently opened and closed, often leaving slots available for other channels to fill. After a source has obtained a slot from the download scheduler, that source is given to the data prioritization scheduler to determine what bytes to download from that source.

[0041] When a source is initially added to the download engine, either at the beginning of the download, or as a new source during a download, there may not be any information available for the Source Ranker to make a decision about the desirability of that source. Since characteristics such as throughput and latency are often only discernable by downloading from the source for a while, the channel scheduler will give new sources a chance to immediately obtain a slot, up to the maximum number of slots.

[0042] Once enough information is available for the Source Ranker to properly evaluate the channel source, it is thereafter required to acquire one of the normal slots, of which there is a limited number that is less than the maximum number of slots. In general, as soon as a normal slot becomes available, the channel scheduler will choose the highest ranked available source and give the slot to that source. As soon as the source has been allocated a slot, it is denoted as no longer be available and is not eligible to obtain a new slot until it finishes its current transfer. However, keep in mind that a single physical source, such as a web server, may be identified by multiple URIs which allows the invention to schedule multiple connections to a single physical source at the same time.

[0043] Another non-limiting example would be to highly prefer nearby sources on a computer network such that if a nearby source is detected, the engine will only schedule from that source even though multiple source slots may be available. In another possible usage, very high priority sources may utilize multiple slots, or fewer slots may become available if very desirable hosts are located.

[0044] Another major feature of the channel scheduler is that it can perform compensate for latency to a source by "pipe-lining" requests to a single source. Thus, the source scheduler may make a source available to be rescheduled slightly before it is finished with its current transfer. This allows the prioritization scheduler to reschedule the source and make a new connection before the current transfer completes. Ideally the new connection will be established exactly when the previous request is complete, thus avoiding wasting any time between transfers.

Prioritization Scheduler

[0045] As illustrated in FIG. 5, after the Channel Scheduler has chosen a source to fill a slot, that source is then passed on to the prioritization scheduler, which will decide what data is requested from the source and at what time. The prioritization scheduler tracks the latency and throughput of all channels at all times and dynamically adjusts its scheduling in order to maximize download performance while still prioritizing the order in which that data will be received.

Data Prioritizer

[0046] The first component in the prioritization scheduler is the data prioritizer, which determines the priority of the data to be scheduled. The data prioritizer specifies an ordering of data from the highest priority data to the lowest priority data. Two examples of policies for the data prioritizer are sequential prioritization and on-demand prioritization.

[0047] Sequential prioritization is a simple and common prioritization where the first byte of the file has the highest priority, the second byte has the next highest, and so on until the last byte which has the lowest priority. For simplicity, most of this specification makes explicit reference to sequential or progressive prioritization, though the invention is not limited to these prioritizations.

[0048] On-demand prioritization specifies an ordering that can change during the lifetime of the download. In on-demand prioritization, the highest priority data is that which external applications are attempting to access, the next highest priority data is the next data the applications is expected to access. In this scheme, the lowest priority data is that which is expected to be accessed last. Another component that can drive the data prioritizer is the corruption repair engine. If the corruption repair engine detects corruption in previously downloaded bytes, it will specifiy that the prioritizer set a higher priority on the bytes that it is repairing rather than new bytes that are being downloaded. In the case where multiple components or applications are attempting to simultaneously specify different portions of the data set to prioritize, the data prioritizer may employ a prioritization scheme such as First Come First Serve, Last Come First Serve, Random, Shortest Processing Time First, Round Robin, or Shortest Remaining Processing Time First or any other flow scheduling algorithm. In the First Come First Serve scheme, the data set portions specified first get the highest priority. In the Last Come First Serve scheme, the data set portions specified most recently get the highest priority. In the Random scheme, the data set portions are chosen at random. In the Shortest Processing Time First scheme, the smallest, or cheapest to retrieve data set portions are given the highest priority. In the Round Robin scheme, each data set portion is made the highest priority data in turn in order to ensure scheduling fairness. In the Shortest

Remaining Processing Time First scheme, the data set portion with the least amount of data waiting to be received is given the highest priority.

[0049] The data order prioritizer may also prioritize the data order based on high-level system or network optimization policies. An example of one such policy is for the data order prioritizor to give the data set portions that are available from only a few sources a higher priority than data set portions that are widely available.

Range Scheduling

[0050] As an optimization, the prioritization scheduler schedules *ranges* of bytes to be downloaded. A range is essentially a run length encoding of a sequential list of bytes. A set of bytes can then be compactly represented as a *range set* which allows all normal mathematical set operations to be efficiently performed on the list of bytes. The invention may utilize range sets extensively in its implementation for any task that requires keeping track of a list of bytes. Examples of ranges and range sets are as follows:

- 10-20 - Inclusive list of bytes 10,11,12,...,20
- 1,3,10-20 – Inclusive list of bytes 1, 3 and 10,11,12,...,20
- 0- - Inclusive list of all bytes from 0 to inifinity, or the end of the file.
- -20- - Negative ranges start from the end of the file. So this range is a list of the last 20 bytes in the file

[0051] Ranges provide a very compact and natural way to describe scheduling of a content channel. Most modern network protocols such as HTTP and FTP support specifying a range of bytes that the receiver desires to download. For example, in HTTP, this is accomplished by using a special header as follows:

```
Range: bytes=100-3000
```

[0052] Additionally, the ranges scheduled by the prioritization scheduler do not necessarily need to correspond to a range of bytes at a given source. These ranges may correspond to data that is interleaved or otherwise encoded. In these cases, a range adapter is used to map the scheduled range onto the list of interleaved or encoded bytes that the source can interpret.

[0053] Also note that a single byte is still a range with one element. So, although much of this specification refers to ranges of bytes, it also encompasses byte-by-byte data transfer and scheduling. Ranges also encompass fixed-size data packets or blocks of data as well. In the cases where this specification refers to a dynamically sized range of data, this also includes a dynamically calculated number of data packets or data blocks.

Proportional Allocator

[0054] After the data has been prioritized, the Proportional Allocator determines how much data will be transferred from the channel that is being scheduled. This amount of data will be proportional to the throughput that is expected to be received over the channel from the source currently being scheduled. If desired, this policy allows multiple channels transferring data in parallel to complete their transfers at roughly the same time. Thus, if there is a source A that has an expected throughput that is ten times faster than source B, the proportional allocator will allocate ten times more data to the schedule for source A. The proportion of data is also bounded by minimum and maximum values, or can be hard coded for a specific amount of data, making the amount of data received equal among all channels.

[0055] In the suggested implementation of the invention, the proportional allocator will use time intervals to determine the proportion of data to be downloaded. First, the proportional allocator, chooses an interval T, and calculates how much data is expected to be received from the source being scheduled based off of its expected throughput and latency. This approach fairly allocates the amount of data among the sources without having to know the throughputs of all of the sources ahead of time. This approach also places a bound on the amount of time data is being received from a channel, allowing other sources an opportunity to be scheduled once the data has been received.

[0056] The proportional allocator automatically responds to different transmission speeds, scaling to provide very low over head data transfer for high speed channels, while ensuring that low speed channels get tight interleaving between the channels and plenty of chances to reschedule channels.

[0057] An enhancement to the interval-based approach is to dynamically change the interval based on the expected overall throughput across all channels. In this case, a given amount of data is scheduled to be allocated during the current interval. The time period T, is then set to be equal to the expected amount of time it will take to download that amount of data from all channels. This approach allows the interval-based allocator to ensure proportional allocation for each block of data within the file.

[0058] Another approach to the proportion allocator is to allocate fixed size and usually small amounts of data to each source. Once the source has received most of the data it has scheduled and is ready to pipeline another request, it is allocated another fixed size amount of data. In this fashion, each source is scheduled proportional amounts of data, because the fast channels will request allocations at a proportionally higher frequency than slow channels. This approach is similar to what happens when the maximum packet size in the interval allocator is set to be very small relative to the size of the interval.

Bulk Scheduler

[0059] After the proportion allocator has determined the amount of data to be received from the source, the bulk scheduler then determines which bytes will be scheduled. The bulk scheduler maintains a list of data that has yet to be scheduled, as well as a list of data that has already been scheduled, but has yet to be received from a channel. If available, the bulk scheduler will schedule the highest priority unscheduled bytes up to the amount of data specified by the proportional allocator. In some cases, this may be less than the allocated number of bytes, in which case the smaller amount of data is actually scheduled.

[0060] If no unscheduled bytes are available, the bulk scheduler will schedule high priority bytes that have been scheduled, but have not yet been downloaded. Also, if the range has already been scheduled, but is just being served by a slow downloader, then it downloads from the end of the undownloaded range, pre-empting some of the range that has already been scheduled for another source. Once the other source catches up to the preempting source, the download from the other source is canceled.

[0061] Any time that the Bulk Scheduler preempts the download of another channel, it attempts to schedule the bytes to be downloaded such that the original source will catch up with the preempting source at the exact moment that the preempting source finishes its transfer.

Advanced Scheduler

[0062] The advanced scheduler provides a number of optimizations over the bulk scheduler. First, the advanced scheduler integrates with the interval-based proportional allocator to allow data to be allocated during a different time interval than the current one. It then figures out how much time is left in the current cycle, then calculates the time needed to complete a range request by setting a minimum range size and incorporating the amount of time necessary to accommodate for the latency of making a new connection. If this download will not be able to be completed within the current cycle, it is scheduled to complete during the next cycle.

[0063] Normally the download is started from the front of the unscheduled ranges, but if the source is too slow to complete the minimum required bytes within the cycle period, then it is required to download from the end of the range. During each new schedule, we check to see which source is the bottleneck in providing the next high priority bytes, if this host has not provided its content within a reasonable time period of the end of the cycle, then the new source is scheduled with the undownloaded bytes of the bottleneck source as the parent for the new download. The advanced scheduler will then schedule the suffix of that range to be downloaded.

Sub-Interval Smoothing

[0064] The advanced scheduler features a sub-interval smoothing component that works to minimize the burstiness of the availability of high priority data with the interval-based allocator. With the interval-based allocator, one source is always going to be the bottleneck for the stream while the other sources provide data that is a bit lower priority. Once the stream is ready to read lower priority data that has already been downloaded by another channel, the already downloaded data is read in a burst until it catches up with the next bottleneck source. In this fashion the priority stream consumption goes from relatively slow consumption to very high consumption as the stream catches up with the already buffered data.

[0065] The invention works to eliminate this burstiness as much as possible to enable the delivery of high priority data in as optimal an order as possible. The sub-interval smoothing works by intelligently interleaving the data from the multiple channels that will be delivering data during the interval. For a given set of data clusters, find the single data cluster that will take the longest amount of time to retrieve. Then locate any clusters that will take equal or less time to retrieve. If multiple clusters from the same source can combine to take less time than the initial cluster, then these multiple clusters will be combined together. Once all clusters have been allocated and found, the clusters are ordered by the fastest source to the slowest source. These clusters can be scheduled incrementally using historical ranking between sources. Thus if one source was the fastest in a previous schedule, it would be expected to be the fastest again in the next schedule and would be scheduled before other clusters in the next interval.

[0066] The result of this sub-interval smoothing is that the fastest source is almost always the bottleneck of the download, and it minimizes the length of the burst when that fast source catches up with the already downloaded data from the other sources. A minimum cluster size may be dictated by latency and throughput to enable efficient pipelining of requests.

Minimized Latency Scheduling

[0067] At the very beginning of the download before any historical data has been established with which to predict future throughput of sources, or perhaps even before the length of the file is known, the invention utilizes special optimizations to reduce the latency of delivering small objects. In general a key goal of the invention is to reduce inefficiency by not downloading redundant information. The minimized latency scheduler sacrifices some of this efficiency for lower latency by requesting the same range of information from many sources in parallel. In general, this amount of information is a small bit of information from the front of the file such that if the total file size is less than that segment size, then the download will be complete as soon as the first channel responds back with data. If the file is larger than the segment size, then the initial request from the sources may introduce some redundant data, but more importantly the scheduler now has performance information on the sources and can use that to optimize the schedule.

[0068] An additional optimization for the case when the file size is unknown is to have every other source request its data from a negative offset, which is from the end of the file, in order to minimize overlap in the case that the file is larger than the size of the blocks being requested from the hosts.

Constraint Scheduler

[0069] Since not all sources have the same capabilities, or even the same data, a constraint scheduler is utilized to ensure that the scheduler does not attempt to retrieve data that the source or channel cannot provide. An example of the constraint scheduler utilizes a list of bytes that the source advertises that it has available. If some of the bytes that the scheduler is trying to schedule for that source are unavailable, the scheduler will choose lower-priority bytes that are available for the scheduler to download.

[0070] Sources are not necessarily required to store an entire copy of the content. Sources may communicate back with the download scheduling engine about the ranges of bytes they have available, possibly using Run-Length Encoding to concisely describe those bytes. Other encodings may be used, including predetermined (or not predetermined) schedules or interleavings of bytes. The scheduling engine will then use the knowledge of which sources have which bytes to determine when to download content from various sources. The range of bytes provided by a given source may also change over time, so that fewer or greater bytes are available from that source. A source may also include a different version of the same file that shares some bytes in common with the version that is intended to be downloaded. These bytes need not correspond to the same byte position in each file and a decoder module would be used to translate between the bytes locations in the different versions of the file.

[0071] If the source is highly constrained in the data it can deliver, this can also negatively affect the source's ranking to the source scheduler.

Channel Connector

[0072] After the prioritization scheduler chooses the range of bytes to be downloaded, it is the channel connector's job to establish a new channel to the scheduled source. The protocol spoken by the channel connector to the channel source is translated by the channel adapters. In general, the channel connector will make a request to the source that contains the range or ranges of information that is scheduled to be downloaded. The time is then measured between when the channel requests the new range and when data starts being received. This time is the latency of the connection. For many protocols, it is common for the channel connector to receive meta-data about the file in response to the connection, such as the file length, media type, and caching information. The Channel Connector then verifies this data with the known meta-data for the file. If there are critical differences between the channel meta-data and the known file meta-data, such as a difference in the file size, then it is assumed that the channel has the incorrect file, or an incorrect version of the file, and the channel connection is then canceled.

[0073] If there is a problem in connecting to the channel, the channel connector will notify the source scheduler of the situation and the source schedule will attempt to another get a slot for the source, reschedule it, and retry the connection process.

[0074] Another thing that can happen is that the source itself may choose to redirect the download request to another source. In this case, the channel connector automatically redirects the channel to the new source and continues the download just as if it had connected to the original source.

[0075] In some cases, the source may also be busy, or may not have available the content that the scheduler intends to receive. In this case, any information about the source such as the bytes that it has available for scheduling is reported back to the source scheduler and constraint scheduler which then attempt to reschedule that source for a later time or a better range of content.

Channel Receiver

[0076] Once a connection to the channel is established, the channel receiver reads the data from the channel and writes it to the I/O dispatcher. In some cases, the actual bytes given back by the channel will differ from the ones that are requested to be scheduled. In this case, if the bytes haven't already been received or are being used for corruption repair, the channel receiver will accept those bytes and write them to their proper locations. If the channel begins overwriting data that has already been downloaded, then the channel receiver may choose to cancel that transfer.

I/O Dispatcher

[0077] The I/O dispatcher takes the bytes being written from all of the channel receivers and provides them to the integrity verifiers and provides the data to the data store for persistent storage.

Integrity Verifiers

[0078] The integrity verifiers are used to ensure that the desired data is received intact and that none of the channels were providing data that is either corrupt or a different version of the content that is desired. Once some of the data has been verified, it is allowed to be accessed via the output stream. If the data is found to be corrupt, then the Corruption Repair engine is notified of this and will work to repair the corruption. A number of approaches is used to implement integrity verifiers, but the most common approach is to use cryptographic hash algorithms to verify that the content was received in tact.

[0079] A simple integrity verifier uses a full file hash, which is the result of applying a cryptographic hash function or checksum algorithm to the entire file. These full file hash algorithms can be calculated incrementally by feeding the bytes of the content into the hash function in order, but the final result of the hash function cannot be obtained until the entire file has been processed. Thus, if the integrity verifier indicates that some corruption is present in the file, it must provide the entire file to the corruption repair engine, because it cannot know what specific bytes of the file are corrupt.

[0080] An improvement upon this approach is to use multiple hash functions that cover different portions of the file and can pinpoint corruption to smaller sections of the file. One approach to this is to use block hashes, where a hash function is applied to fixed length blocks of the content. This allows the integrity verifier to pinpoint which blocks are corrupt, and any blocks that are successfully verified can be immediately accessed via the output stream.

[0081] A third and suggested implementation of an integrity verifier utilizes Merkle Hash Trees. The Merkle Hash Tree, invented by Ralph Merkle, is a hash construct that has very nice properties for verifying the integrity of files and file subranges in an incremental or out-of-order fashion. This approach has the desired characteristics missing from the full file hash approach and works well for very large files. The idea is to break the file up into a number of small pieces, hash those pieces, and then iteratively combine and rehash the resulting hashes in a tree-like fashion until a single "root hash" is created.

[0082] The root hash by itself behaves exactly the same way that full file hashes do. If the root hash is retrieved from a trusted source, it can be used to verify the integrity of the entire content. More importantly, the root hash can be combined with a small number of other hashes to verify the integrity of any of the file segments. By using more or less data from the hash tree, the verification resolution can be increased or decreased respectively. The suggested implementation of a merkle hash tree verification engine for the invention dynamically adjusts the verification resolution of the tree based on the rate at which the content is currently being downloaded. The dynamic verifier fixes a time interval T and aims to do one verification within each time interval T. In order to accomplish this, it calculates how much data is expected to be downloaded in time T and downloads just enough hash data so that the verification resolution enables verifying all of the content downloaded during T with a single hash. For efficiency, the tree hash is implemented using a stack-based approach that only requires O(Log(n)) hashes to be stored in memory for any given operation, including streaming verification.

[0083] Another important aspect of the hash tree verifier is how it responds in the case that corruption is detected. In this case, the verification resolution will dynamically be increased by one and additional hash data will be retrieved to enable finer-grained integrity checking. The corrupt data block will then be rechecked at the smaller resolution at which point some of the data that was previously thought to be corrupt will be proven to be valid. This process continues until the verification resolution has reached its maximum configured level. In turn, the blocks that are still found to be corrupt at the end of this process will be passed on to the corruption repair engine.

[0084] The Integrity Verification Engine allows multiple content checking algorithms to be combined. An additional component that may be combined with the above cryptographic integrity verifiers is one that uses a known number of bytes, such as the first 256 bytes of a file that it received out of band and uses that to compare against the file being downloaded. Thus the exact matching verifier could be used at the same time as a hash tree or full file hash integrity checking component. If the exact matching verifier matches a block of data successfully, then that data is declared to be valid and does not need to be further verified by other verifiers.

Corruption Repair Engine

[0085] The corruption repair engine has the task of attempting to repair corruption that is detected by the integrity verification engine, and will attempt to prove which source or sources caused the corruption. The integrity verification engine provides the corruption repair engine with a block of data that is corrupt. At the same time, the integrity verification engine may be using additional techniques to try to pinpoint the corruption to a smaller block of data. If a smaller area is pinpointed, then the corruption repair engine will be updated with this information to make it easier to repair the corruption.

[0086] The corruption repair engine keeps track of a list of all of the bytes that have been received from each source. This list may utilize run-length encoded data structures in order to efficiently keep track of various ranges of bytes that have been retrieved. The first step in the corruption repair engine is to determine which sources provided data to the corrupt block. At this point there are a number of possible corruption repair algorithms.

[0087] A recommended corruption repair algorithm is to choose one or more "suspect" sources in each phase of the repair. The Corruption Repair engine then creates a new Source Scheduler that is limited to the non-suspect sources, and creates a new Prioritization Scheduler that is specified to only download the data that is being repaired. A new corruption repair scheduler may be created to repair the corrupt portion while the normal schedule continues with the file download. The data that was received by these suspect sources is then re-downloaded by non-suspect sources, and the integrity verification engine rechecks the integrity of this repair attempt. If the repair attempt is successful, then at least one of the suspects was providing corrupt data. If the repair was not successful, then at least one of the non-suspects was providing corrupt data.

[0088] The choice of suspects can be based on a number of heuristics such as trust metrics, geographical location, software version, etc. One good heuristic is to sort the possible suspects according to how much data was received from them in the corrupt block. The sources that uploaded the least amount of data are then marked as the most likely suspects. This approach is good for two reasons – first it forces corruptors to provide a lot of data in order to avoid being flagged as a suspect. This contrasts to corruptors being able to provide single bytes of corrupt data for very low cost. Secondly, by flagging the ones with the least amount of data, it gives the repair engine a chance to repair the corruption with downloading the least amount of data.

[0089] The suspect choosing algorithm is combinatorial. First single suspects are chosen, then they are combined with other suspects in the case that multiple hosts might be colluding to corrupt the download. The combinatorial algorithm uses most of the previous suspects when generating new suspect combinations so as to minimize the amount of data that needs to be redownloaded during each phase.

[0090] An additional enhancement to this corruption repair process is to keep track of whether or not newly downloaded data is the same as the data that is already downloaded. In this way, it can be probabilistically determined which hosts may be causing the corruption by observing which combinations of hosts are providing different data from each other.

[0091] Download sources are shared across all schedulers so that if a download source is found to be providing corrupt data, then that source is signaled as a bad source and will be removed from all source schedulers. Optionally the corruptor may be reported to other hosts in the network or to a central authority so that other nodes may decide to avoid the corrupting host in the future.

Data Store Filters

[0092] The invention supports arbitrary layering of file storage technologies, to allow things such as keeping track of the bytes downloaded for enabling resumes in the future. One such layered file storage filter would be a compressor or encryptor module that stores the bytes on disk in encrypted or compressed form even though the output stream is providing the content in an uncompressed or unencrypted form.

[0093] Another optional layer is one that allows the physical location of the file store on disk to change while the download is taking place. For instance, if the file store runs out of disk space on one hard drive, it can be transparently migrated to another hard disk without affecting the modules that are reading and writing data to the data store.

[0094] Another possible filter could automatically compress the files in the store or delete old files using a Least Recently Used policy if the file store has become to large or has run out of disk space. This allows the file store to automatically make room for the new download without affecting the download in progress.

Data Store

[0095] The data store physically stores the data on disk as it is being downloaded. This data is then accessed by the output stream and the verification engines in order to verify and process the data. The data from the data store can also be saved as a normal file once the download completes, allowing the content to be accessed as a normal file in a computer file system.

[0096] During the download the file is stored in a temporary location such as a user-specific temporary directory or a system wide temporary directory. Via the control API, it is possible to set the destination where this file will be saved to. As soon as the destination is known, the file is moved to the same directory as the final destination, but still using a temporary file name. The file is not moved to its final destination name until the download has completed entirely and the integrity of the file has been fully verified.

Aggregate Behavior

[0097] Certain data structures can be shared across instances of the invention, such as information about the latency and speed of individual sources, as well as whether or not they have been detected as providing corrupt data in the past. These data structures can be used to quickly determine optimal sources for a separate download that is contacting the same set of hosts.

[0098] The invention can also perform prioritization and queuing across instances. This allows different priorities to be assigned to different content. One such policy is a "smallest content first" policy, whereby small files are scheduled to download before big files.

[0099] One suggested way to implement prioritization is to use shared data structures for bandwidth throttling across all instances of the invention. This allows fine-grained control over the rate that the data is being received from the source.

[0100] In one embodiment, the invention is directed to a method comprising communicating data over a computer network via multiple channels in parallel with data order prioritization. The data order may be prioritized in a sequential or progressive fashion, providing a parallel streaming download. The data order may be prioritized on-demand by external components or internal components such as a corruption repair component. The data order may be prioritized utilizing an algorithm that proportionally allocates the amount of data to be received based on the throughput and latency of the channel or channel source. The data order may be prioritized utilizing an algorithm that uses time-based intervals to proportionally allocate the amount of data to be received from a channel during the interval. The data order may be prioritized utilizing an algorithm that tends to make the fastest channel the bottleneck for the prioritized data transfer. The data order may be prioritized utilizing an algorithm that preempts bottleneck channels. The rate of the prioritized data may be smoothed to minimize burstiness of the transfer. The data transmission may take place over a peer-to-peer network. The multiple channels may terminate at the same source. The channels may be heterogeneous, with different protocols and channel adapters used to receive data from these multiple channels in parallel. The channels may be chosen by a ranking of channels or channel sources. New channels and channel sources may be added dynamically during the data transfer. The channels may have adapters that provide encryption, compression, or delta encoding.

[0101] In another embodiment, a method comprises providing integrity verification and corruption repair for the data transfer. The data may be transferred over multiple channels in parallel with data order prioritization. The integrity verification may utilize an iterative hash construct, such as a Merkle Hash Tree, and a corruption repair engine utilizes combinations of channel sources to pinpoint corruption. An amount of data that a source has provided may be used in determining the likeliness that the source caused the corruption. A majority of sources may remain the same between iterations. Integrity checking and corruption repair may be performed in iterations. The corruption can be further pinpointed by increasing the resolution of the verification algorithm or by employing multiple verification algorithms. A source may be proven to have caused the corruption, and may be announced to other components in the system or external systems. Bandwidth throttling can be employed to prioritize or deprioritize channels for scheduling. The bandwidth throttling is used to slow the overall data transfer across all channels. The bandwidth throttling can be used across all channels to the source to limit the data transfer rate to that source. A highly ranked source can be pipelined to compensate for latency. Slots can be used to track which sources are currently scheduled.

[0102] In another embodiment, a computer-readable medium comprises instructions to cause a processor to receive data over multiple channels in parallel with data order prioritization and present the data to a user.

[0103] Various embodiments have been described. The described techniques can be embodied in a variety of devices, including personal computers, workstations, servers, mobile phones, laptop computers, handheld computing devices, personal digital assistants (PDA's), and the like. The devices may include a microprocessor, a digital signal processor (DSP), field programmable gate array (FPGA), application specific integrated circuit (ASIC) or similar hardware, firmware and/or software for implementing the techniques. If implemented in software, a computer-readable medium may store computer readable instructions, i.e., program code, that can be executed by a processor to carry out one of more of the techniques described above. For example, the computer-readable medium may comprise random access memory (RAM), read-only memory (ROM), non-volatile random access memory (NVRAM), electrically erasable programmable read-only memory (EEPROM), flash memory, or the like. These and other embodiments are within the scope of the following claims.